# ADVANCED DISTRIBUTED SIMULATION TECHNOLOGY



## ModSAF

## PROGRAMMER'S REFERENCE

## MANUAL

## VOL. 4

**(Libuoverwatchmove- Libxfile)**

Ver 1.0 - 20 December 1993

CONTRACT NO. N61339-91-D-0001

D.O.: 0021

CDRL SEQUENCE NO. A001



Prepared for:

U.S. Army Simulation, Training, and Instrumentation Command (STRICOM)
12350 Research Parkway
Orlando, FL 32826-3276

Prepared by:

LORAL Systems Company

ADST Program Office
12151-A Research Parkway
Orlando, FL 32826

# ADVANCED DISTRIBUTED SIMULATION TECHNOLOGY

## ModSAF

## PROGRAMMER'S REFERENCE

## MANUAL

## VOL. 4

**(Libuoverwatchmove- Libxfile)**

Ver 1.0 - 20 December 1993

CONTRACT NO. N61339-91-D-0001

D.O.: 0021

CDRL SEQUENCE NO. A001



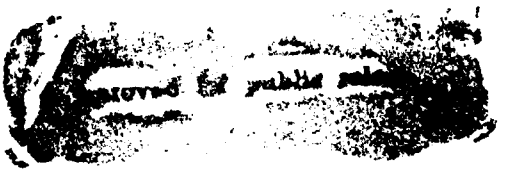
Prepared for:

U.S. Army Simulation, Training, and Instrumentation Command (STRICOM)
12350 Research Parkway
Orlando, FL 32826-3276

Prepared by:

LORAL
Systems Company

ADST Program Office
12151-A Research Parkway
Orlando, FL 32826



**Libuoverwatchmove**

# Table of Contents

# 1 Overview

TEMPLATE: Describe what this library does here.

## 1.1 Examples

TEMPLATE: Give examples here.

# 2 Functions

The following sections describe each function provided by libuoverwatchmove, including the format and meaning of its arguments, and the meaning of its return values (if any).

TEMPLATE: Adjust alignment of descriptions

TEMPLATE: Correct argument lists and descriptions of these functions.

## 2.1 uowm_init

```
void uowm_init()
```

uowm_init initializes libuoverwatchmove. Call this before any other libuoverwatchmove function.

## 2.2 uowm_class_init

```
void uowm_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Class of the parent (declared with class_declare_class)

uowm_class_init creates a handle for attaching uoverwatchmove class information to vehicles. The parent_class will likely be safobj_class.

## 2.3 uowm_create

```
void uowm_create(vehicle_id, params, po_db, ctdb, quad_data)
    int                     vehicle_id;
    UOWM_PARAMETRIC_DATA    *params;
    PO_DATABASE             *po_db;
    CTDB                    *ctdb;
    QUAD_DATA               *quad_data;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'po_db' Specifies the PO database

'ctdb, quad_data'
Specify the terrain database

uowm_create creates the uoverwatchmove class information for a vehicle and attaches it vehicle's block of libclass user data.

## 2.4 uowm_destroy

```
void uowm_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

uowm_destroy frees the uoverwatchmove class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

**Libupoccpos**

# Table of Contents

# 1 Overview

Libupoccpos implements a unit-level Preparatory task for Occupy Position. This preparatory task finds covered and/or concealed positions along the battle position and instructs the subordinates to go toward these positions. This task ends when the subordinates have reached the desired positions. The task state machine is written using the AAFSM format which is translated to C using the '**fsm2ch**' utility (see section 'Overview' in LibTask Programmer's Manual).

Four parameters are passed to **SM_UPrepOcpyPos**: left TRP, right TRP, engagement area TRP, and battle position. Based on the battle position and the number of subordinates, the number of vehicles per segment (the battle position is made up of one or more line segments) and the battle areas (areas where each vehicle searches for cover) are calculated. The subordinates are ordered by job numbers, and are assigned positions from one end of the battle position to the other end so no vehicle crossover occurs while they are traveling to their positions. The cover/concealment searching algorithm uses ctdb utilities which can tend to be expensive, especially is the battle areas are large. Therefore, the search is divided among several ticks. Once the positions are found, they are passed to vmove for each vehicle. The state machine then sits and waits until the vehicles are in their desired positions.

## 1.1 Task Parameters

The **SM_UPrepOcpyPos** task has four parameters:

```
typedef struct upoccpos_parameters
{
    ObjectID engagement_area;
    ObjectID trp_right;
    ObjectID trp_left;
    ObjectID battle_position;

} UPOCCPOS_PARAMETERS;
```

**'engagement_area'**
: Specifies a persistent object which defines the engagement area. The engagement area is a point object.

**'trp_right'**
: Specifies a persistent object which defines the right bound of the sector of fire for the unit. This object is a point object.

'trp_left'
Specifies a persistent object which defines the left bound of the sector of fire for the unit. This object is a point object.

'battle_position'
Specifies a persistent object which defines the battle position. The battle position is a line object.

## 1.2 Parametric Data

There are eleven parametric data entries for SM_UPrepOcpyPos:

**min_allowable_visibility** has a value from 0.0 to 1.0 and specifies the minimum visibility (0.0 being not visible and 1.0 being completely visible) of the engagement area TRP required for a point to be considered a good cover position

**tree_opacity** has a value from 0.0 to 1.0 and specifies the degree to which one can see through tree lines. 0.0 means tree lines are ignored, and 1.0 means that tree lines completely block the path of vision

**front_dist_percentage** specifies how far in front of the battle position to search for cover positions. The value specified is a percentage of the total battle position length.

**back_dist_percentage** specifies how far back from the battle position to search for cover positions. The value specified is a percentage of the total battle position length.

**fire_sector_overlap_percentage** specifies the degree to which fire sectors overlap

**grid_spacing** specifies how carefully the search will be performed. A low value specifies a quick search (possibly overlooking some good cover positions) while a high value specifies a search that takes longer but detects more cover positions. This value is in meters and specifies the distance between sample points.

**searches_per_tick** specifies how many sample points for each vehicle will be processed per tick. If there are four vehicles and two searches per tick, then eight sample points will be processed per tick.

**max_building_width** is used to ensure that a concealed point doesn't end up inside a building.

It should be set to the maximum building width in the area being searched.

**hull_coverage** specifies the percentage of the hull that is protected from direct fire by the ground. It is a number between 0.0 and 1.0.

**engagement_area_size** specifies the size of the engagement area. It is expressed in percentage meters. The front of the engagement area is a certain distance in front of the engagement area TRP. This distance is a percentage of the distance from the battle position to the engagement area TRP. **engagement_area_size** specifies this percentage. Anything closer to the battle position than the front of the engagement area TRP will be blocked from view by the earth.

**speed** specifies the speed (in $m/s$) at which ground vehicles will move to their positions

# 2 Functions

The following sections describe each function provided by libupoccpos, including the format and meaning of its arguments, and the meaning of its return values (if any).

## 2.1 upoccpos_init

```
void upoccpos_init()
```

upoccpos_init initializes libupoccpos. Call this before any other libupoccpos function.

## 2.2 upoccpos_class_init

```
void upoccpos_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Class of the parent (declared with class_declare_class)

upoccpos_class_init creates a handle for attaching upoccpos class information to vehicles. The parent_class will likely be safobj_class.

## 2.3 upoccpos_create

```
void upoccpos_create(vehicle_id, params, po_db, ctdb)
    int32                        vehicle_id;
    UPOCCPOS_PARAMETRIC_DATA     *params;
    PO_DATABASE                  *po_db;
    CTDB                         *ctdb;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'po_db' Specifies the PO database where the task can be found

'ctdb' Specifies the terrain database currently in use

upoccpos_create creates the upoccpos class information for a vehicle and attaches it vehicle's block of libclass user data.

## 2.4 upoccpos_destroy

```
void upoccpos_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

upoccpos_destroy frees the upoccpos class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

## 2.5 upoccpos_init_task_state

```
void upoccpos_init_task_state(task, state)
    TaskClass      *task;
    TaskStateClass *state;
```

'task' Specifies a pointer to the task class object to be initialized.

'state' Returns the initialized state

Given a new SM_UPrepOcpyPos task that is about to be created, upoccpos_init_task_state initializes the model size, and state variables.

# 3 Algorithms

## 3.1 Cover Algorithm

Instead of finding the cover positions in one tick, the search is distributed among several ticks until good cover positions are found. Each vehicle is given an area about the battle position which to search, which will be referred to as the 'battle area'. Also, a default position along battle front (which the vehicle will use as its destination point if no cover or concealment is found) is calculated for each vehicle. For each vehicle, three 'lines' consisting of equally spaced points are constructed: one along the left side of the battle area, one along the back, and one along the right side. During each tick, a few of these points are used as "starting points" or "sample points" in the search for cover positions. For each sample point, a profile array (3-dimensional) which contains the elevation of the ground along a vector from the sample point to the engagement area TRP is generated. Any two consecutive entries in this array define the endpoints of a segment along the profile with a constant slope.

Then, starting with the segment closest to the engagement area TRP, and progressing to the last segment (ending at the sample point), the following occurs:

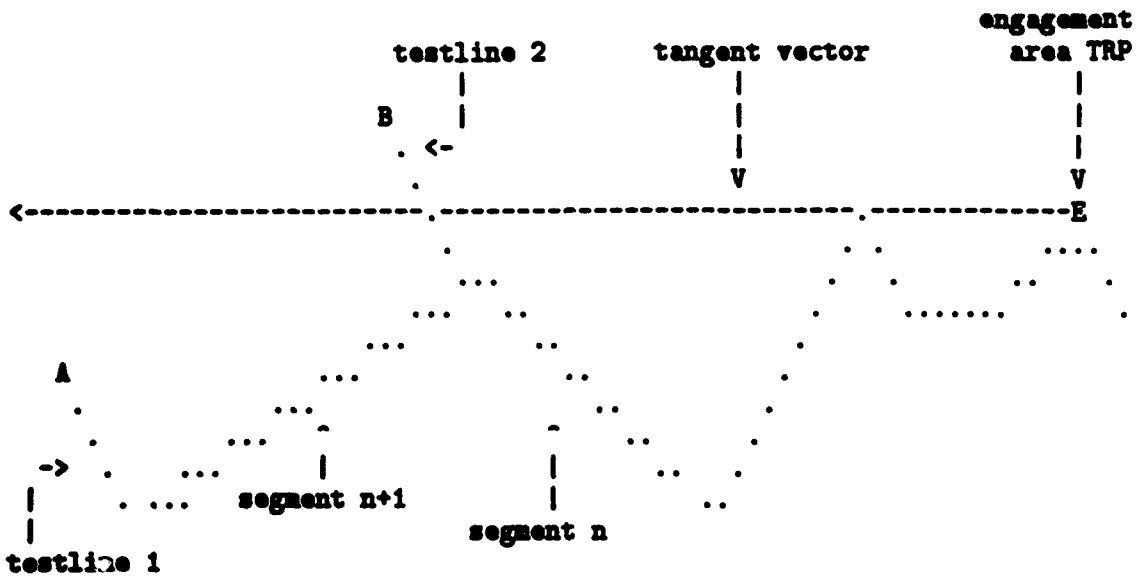
1) The maximum slope of the line from the engagement area TRP to the point of the segment closest to the engagement area TRP is kept track of. This is the line of sight of the enemy (also known as the 'tangent vector.')

2) Two line segments are generated, each starting from one end of the current profile segment. These line segments are perpendicular to the ground at the current profile segment and have a length that is equal to the body height of the vehicle that is searching for cover. The line segment closest to the sample point will be referred to as 'testline 1', and that closest to the engagement area TRP as 'testline 2'.

3) If the tangent vector does not intersect with testline 1 but does intersect with testline 2, then there is a hull defilade position somewhere along that profile segment. The exact point is calculated by determining the intersection of the line whose endpoints are the top ends of the testlines (points A and B on the diagram) with the tangent vector, then projecting onto the ground.

For each vehicle, the covered positions with the mildest slope is used as the destination point.

In the following diagram, a hull defilade position is found to be somewhere on segment n+1.




## 3.2 Concealment Algorithm

Concealed points are only searched for if no cover points are found. ctdb_find_ground_intersection() is used to determine if there is a treeline or building along certain segments of the profile array. If so, the location is marked as a concealed position.

Not all parts of all segments are searched for concealment. Only the portions of segments that are above (higher in altitude) the tangent line are searched, since intervisibility would fail for any portion of terrain below the tangent line. Therefore, segments that have the tangent line pass above them are not checked at all for concealed positions.

The concealed positions that are closest to the vehicles' default locations on the battle position are used as the destination points. Concealed positions are only used if no cover positions are found.

# 4 Debugging

When debugging for Prep Occupy Position is enabled for any vehicle(s) (via 'veh X debug upoccpos on'), three different colors of TRP-style points are put in each vehicle's overlay: yellow, red, and green.

**The yellow points represent the sample points (see the Algorithms section) for a given vehicle.**

**The red points represent the cover positions found.**

**The green points represent the concealed positions found.**

# LibURTB

# Table of Contents

# 1 Overview

Liburtb implements a unit-level task which controls a group (currently only one) of vehicles returning to a base and landing. The task state machine is written using the AAFSM format which is translated to C using the 'fsm2ch' utility (see section 'Overview' in LibTask Programmer's Manual).

Liburtb depends on libvtakeoff, libvflwrte, libvland, libpo, libvtab, libclass, libctdb, libaccess, libreader, and libparmgr.

## 1.1 Task Parameters

When a SM_URTB task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are represented in the task data structure as follows:

```
typedef struct urtb_parameters
{
    ObjectID base;
    uint16   padding;
    float64  speed;
    float64  altitude;
} URTB_PARAMETERS;
```

'base' Specifies the base for the vehicle to return to. This base can be a point object, line object, or a text object.

'speed' Specifies the speed of the vehicle.

'altitude' Specifies the altitude of the vehicle.

# 2 Functions

The following sections describe each function provided by liburtb, including the format and meaning of its arguments, and the meaning of its return values (if any).

## 2.1 urtb_init

```
void urtb_init()
```

urtb_init initializes liburtb. Call this before any other liburtb function.

## 2.2 urtb_class_init

```
void urtb_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Class of the parent (declared with class_declare_class)

urtb_class_init creates a handle for attaching urtb class information to vehicles. The parent_class will likely be safobj_class.

## 2.3 urtb_create

```
void urtb_create(vehicle_id, params, po_db, ctdb)
    int32   vehicle_id;
    URTB_PARAMETRIC_DATA *params;
    PO_DATABASE          *po_db;
    CTDB                 *ctdb;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'po_db' Specifies the PO database where the task can be found

'ctdb' Specifies the terrain database currently in use

urtb_create creates the urtb class information for a vehicle and attaches it vehicle's block of libclass user data.

## 2.4 urtb_destroy

```
void urtb_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

urtb_destroy frees the urtb class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

## 2.5 urtb_init_task_state

```
void urtb_init_task_state(task, state)
    TaskClass *task;
    TaskStateClass *state;
```

'task' Specifies a pointer to the task class object to be initialized.

'state' Returns the initialized state

Given a new SM_URTB task that is about to be created, urtb_init_task_state initializes the model size, and state variables.

# Libutargeter

# Table of Contents

# 1 Overview

Libutargeter implements a unit level task that gets a list of all capable vehicles in that unit. If the fire technique is alternating then the capable vehicles will get paired up before giving vtargeter the appropriate targeting information. The task state machine is written using the AAFSM format which is translated to C using the 'fsm2ch' utility (see section 'Overview' in LibTask Programmer's Manual).

During targeting operations, there are two states as described below

'start' When in this state, the task gets the number of capable vehicles and spawns the vehicle targeting task libvtargeter. If the fire technique is set to alternating then the vehicles are paired up before spawning the vehicle targeting task.

'monitoring'
When in this state, the task gets the number of capable vehicles and spawns the vehicle targeting task libvtargeter. If the fire technique is set to alternating then the vehicles are paired up before spawning the vehicle targeting task.

The types of recommendations made by this task during either state can be controlled by the parametric data for this unit subclass, as described below.

## 1.1 Parametric Data

The format of the parametric data for this unit subclass is as follows:

```
(SM_UTargeter (firing_pause <integer milliseconds>))
```

The firing_pause designates the amount of time after the coordinated vehicle fires before the paired vehicle can fire. This is passed to vtargeter.

## 1.2 Task Parameters

LibUtargeter uses task parameters only when configured for ground. The parameters ared described by the following structures.

```
typedef enum vtargeter_fire_permission
{
    VTARGETER_WEAPONS_HOLD  = 1,
    VTARGETER_WEAPONS_FREE  = 2,
    VTARGETER_WEAPONS_TIGHT = 3,
} VTARGETER_FIRE_PERMISSION;


typedef enum Vtargeter_fire_technique
{
    VTARGETER_FIRE_TECHNIQUE_SIMULTANEOUS = 1,
    VTARGETER_FIRE_TECHNIQUE_ALTERNATING  = 2,
} VTARGETER_FIRE_TECHNIQUE;

typedef struct utargeter_parameters
{
    ObjectID                  vsctr_rgt_bnd[UNITORG_MAX_BREADTH];
    ObjectID                  vsctr_lft_bnd[UNITORG_MAX_BREADTH];
    VTARGETER_FIRE_PERMISSION permission;
    VTARGETER_FIRE_TECHNIQUE  vtarg_fire_technique;
    VASSESS_MODE              vass_mode;
    float32                   range;
    float32                   fire_at_pos[2];
    int32                     num_point_sets;
    VASSESS_FIRE_TYPE         fire_type;
} UTARGETER_PARAMETERS;
```

**vsctr_rgt_bnd**

**vsctr_lft_bnd**

**permission** specifies whether the vehicle cannot fire, can fire at will, or can only fire when fired upon.

**vtarg_fire_technique** specifies whether the vehicle can fire when it wants, or should wait to fire after another vehicle fires (for alternating fire).

**vass_mode** specifies the method used to determine targets. The three modes are "Closest to Self", "Sector Points", and "Closest to Location".

**range** specifies the maximum distance that the vehicle can shoot.

**fire_at_pos** is the target location.

**num_point_sets** is not used.

**fire_type** specifies the method of firing at the enemy. The three types are "Distributed Fire", "Volley Fire", and "None".

# 2 Functions

The following sections describe each function provided by libutargeter, including the format and meaning of its arguments, and the meaning of its return values (if any).

## 2.1 utarg_init

```
void utarg_init()
```

utarg_init initializes libutargeter. Call this before any other libutargeter function.

## 2.2 utarg_class_init

```
void utarg_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Class of the parent (declared with class_declare_class)

utarg_class_init creates a handle for attaching utargeter class information to vehicles. The parent_class is one created with class_declare_class.

## 2.3 utarg_create

```
void utarg_create(vehicle_id, params, po_db)
    int                          vehicle_id;
    UTARGETER_PARAMETRIC_DATA *params;
    PO_DATABASE                 *po_db;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'po_db' Specifies a PO database where task objects can be located

utarg_create creates the utargeter class information for a vehicle and attaches it vehicle's block of libclass user data.

## 2.4 utarg_destroy

```
void utarg_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

utarg_destroy frees the utargeter class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

## 2.5 utarg_init_task_state

```
void utarg_init_task_state(task, state)
    TaskClass *task;
    TaskStateClass *state;
```

'task' Specifies a pointer to the task class object to be initialized.

'state' Returns the initialized state

Given a new SM_Utargeter task that is about to be created, utarg_init_task_state initializes the model size, and state variables.

**Libutraveling**

**Libutraveling**

# Table of Contents

# 1 Overview

The unit traveling library provides a method for tasking a unit of vehicles to move along a route. The route is specified via a `PointClass`, `TextClass`, or `LineClass` PO object. If the unit is larger than one vehicle, the vehicles in the unit will be kept in formation by increasing or decreasing their command speeds periodically.

This library should eventually handle arbitrarily sized units in a hierarchical manner. At this time, it will try to control all the vehicles in the specified unit directly.

## 1.1 Parameters

```
ObjectID    route;
float64     speed;
float64     speed_limit;
uint8       formation[];
uint8       form_type;
uint8       roadmarch;
uint8       conform;
```

**'route'** The route is a `LineClass`, `PointClass`, or `TextClass` PO. It provides the general route that the unit will follow.

**'speed'** The speed defines the speed of the unit. If there is only one vehicle in the unit, it will travel at exactly this speed. If there is more than one, the speed is used to compute the speed of the individual vehicles to keep them in formation.

**'speed_limit'**
This somewhat poorly named parameter defines the maximum speed that the vehicles may travel. It should be zero if no speed limit is desired, or some number higher than the speed parameter. If defined, this will be the speed used as the catch-up speed.

**'formation'**
A character string defining a formation name understood by `libformationdb`.

**'form_type'**
Another poorly named parameter. Either open (0) or closed (1). This is currently only used for roadmarch spacing. The distances meant by open and closed is defined in the parametric data as `open_spacing` and `closed_spacing`.

**'roadmarch'**
A boolean value that determines whether roads will be used in the route or if formations

will proceed along the sides of the road. If 1, the vehicles will get on the road and travel in a column in roadmarch order.

'conform' A boolean value that determines whether the input route will be conformed to the terrain. In the future, this will be a multi-value parameter that determines whether to follow valleys or ridgelines or to not conform.

## 1.2 Algorithm

When utraveling is spawned, its input route object undergoes some initial massaging. First, if the route is a single point, it is turned into a two-point route starting at the unit location. Next, if the route is to be conformed to the terrain, a routine in `libroute` is called to change the route. This currently doesn't work particularly well. Finally (not implemented yet) long sections of route will be broken into shorter pieces.

All internal route structures are maintained in the `libroute` format defined in `stdroute.h`. The route is stored in sections composed of cross-country and road segments. If the roadmarch input parameter is turned on, vehicles will drive in an order defined by `libformationdb` on any road sections in the input unit route.

For cross-country sections, libutraveling calls `libformationdb` to split the unit route into individual vehicle routes. These routes will have vertices offset from the unit route vertices based on the input formation parameter.

A routine is then called that takes the list of subordinate vehicles and the individual vehicle routes, and determines a subsections of these routes (somewhat longer than 500 meters). Arrival times to the end of each subsection are computed, and appropriate speeds for each vehicle are determined based on the input unit speed and the distance each vehicle will need to travel to arrive at the endpoint of the subsection at the same time as the other vehicles arrive at their endpoints.

Individual vehicle vmove tasks are then spawned, and given appropriate parameters. In a somewhat atomic operation, the vehicle routes are turned into `LineClass` objects, which are used as input object references. The state of each vmove task is then monitored. In the simplest case, when all vmove tasks report they are in `arrived` state, the `utraveling` task cleans up and goes to `END` state.

Typically, the route is somewhat longer than the subroute passed to the individual vmove tasks, and the vehicles do not behave at all as expected. Thus, much of `utraveling` is devoted to handling silly exception conditions and idiosyncracies of the vmove and movemap libraries.

One of the first things that needs to be dealt with is the length of routes. Road segments and terrain-conformed route sections may cause the massaged vehicle routes to be significantly longer than can be stored in a `LineClass` object. To get around this, `utraveling` breaks the long unit route into short sections. This was also done so that `movemap` could be given reasonably short routes. This is important so that when `movemap` computes a speed at which to travel, it will be reasonable enough to keep the unit roughly in formation.

When each vehicle gets within 500 meters of the end of its subroute, its `vmove` goes to `arriving` state. When all the vehicles are in this state, utraveling computes a new set of subroutes, new arrival times, and passes these as parameters to the `vmoves`.

Since things always happen to cause the vehicles to get out of sync, `utraveling` will periodically (currently every 10 seconds) recompute the arrival times and update the `vmoves`. This helps keep the formations correct.

# 2 Functions

The following sections describe each function provided by libutraveling, including the format and meaning of its arguments, and the meaning of its return values (if any).

TEMPLATE: Adjust alignment of descriptions

## 2.1 utrav_init

```
void utrav_init(routemap)
    ROUTEMAP_PTR routemap;
```

'routemap'
Specifies [illegible] map to use for planning around rivers, canopies, etc.

utrav_init initializes libutraveling. Call this before any other libutraveling function.

## 2.2 utrav_class_init

```
void utrav_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Class of the parent (declared with class_declare_class)

utrav_class_init creates a handle for attaching utraveling class information to vehicles. The parent_class will likely be safobj_class.

## 2.3 utrav_create

```
void utrav_create(vehicle_id, params, po_db, ctdb, quad_data)
    int                          vehicle_id;
    UTRAVELING_PARAMETRIC_DATA   *params;
    PO_DATABASE                  *po_db;
    CTDB                         *ctdb;
```

```
QUAD_DATA                    *quad_data;
```

'vehicle_id'
: Specifies the vehicle ID

'params'
: Specifies initial parameter values

'po_db'
: Specifies the PO database

'ctdb'
: Pointer to the CTDB terrain database

'quad_data'
: Pointer to the quadtree feature database

utrav_create creates the utraveling class information for a vehicle and attaches it vehicle's block of libclass user data.

## 2.4 utrav_destroy

```
void utrav_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
: Specifies the vehicle ID

utrav_destroy frees the utraveling class information for a vehicle. This should be called before freeing the class user data with class_free_user_data.

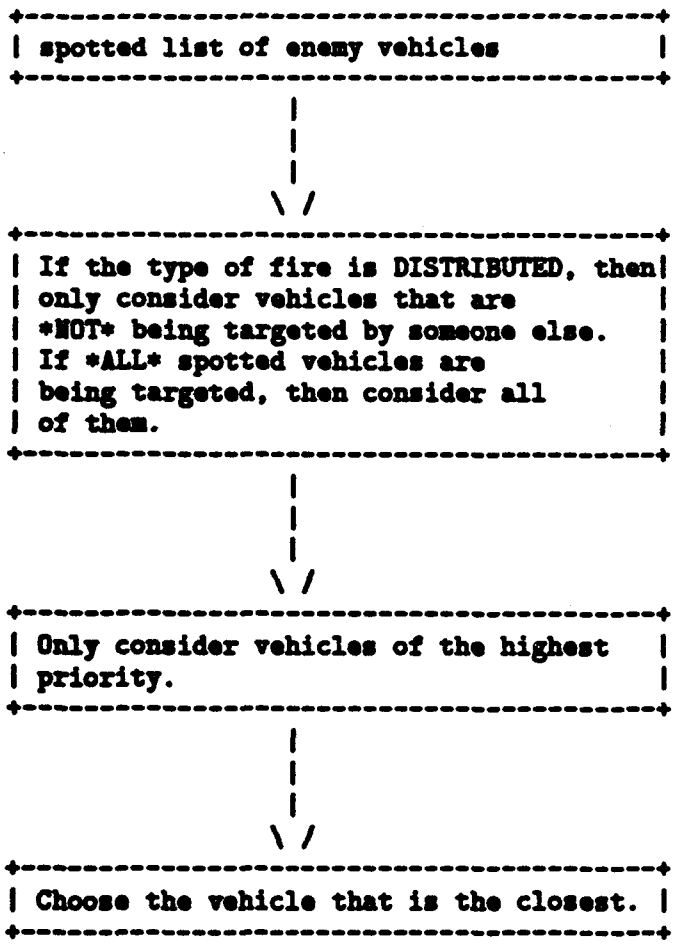
# Libvassess

# Table of Contents

# 1 Overview

Libvassess implements a vehicle level task which identifies the most urgent target and makes recommendations for weapons to use against that target. The task takes its primary input from the libspotter task which provides lists of IFF identified vehicles detected by available sensors. The task state machine is written using the AAFSM format which is translated to C using the 'fsm2ch' utility (see section 'Overview' in LibTask Programmer's Manual).

This diagram gives an overall description of how VAssess chooses a new target.




The format of the parametric data is as follows:

```
(SM_VAssess (background [on | off])
```

```
(sensors <name> <name> ...)
(weapons (<vehicle-class-mask> <vehicle-class-value>
          (<min range> <max range> <name> <munition>)
          (<min range> <max range> <name> <munition>)
          ...)
         (<vehicle-class>
          (<min range> <max range> <name> <munition>)
          (<min range> <max range> <name> <munition>)
          ...)
         ...)
(looking_tick_period <integer msec>)
(reevaluating_tick_period <integer msec>)
(recommendation_persistence_time <integer msec)
(assessment_type [air | ground])
(range_factor <real>)
(aspect_factor <real>)
(persistence_factor <real>)
(mobility_and_fire_factor <real>)
(fire_factor <real>)
(max_hits_on_target <integer>)
(target_priorities
    ((<priority> <vehicle-class-mask> <vehicle-class-value>)
     ...
     (<priority> <vehicle-class-mask> <vehicle-class-value>)
     ...
    ))
)
```

The **background** parameter specifies whether the task should automatically be included in the background task frame of this vehicle. Normally, this will have the value on to indicate it should be included.

The **sensors** parameter contains the names of sensors which may be manipulated during the assessment process. Currently no sensors are manipulated.

The **weapons** parameter contains the list of data used to provide weapon and munition recommendation against threats. **<vehicle-class-mask>** is a bitwise OR combination of SIMNET object type mask fields which will be checked for a match against the values encoded in the **<vehicle-class-value>**. **<vehicle-class-value>** is a bitwise OR combination of SIMNET object type value fields which are used to match against the assumed object type of the threat. If the fields match, the first weapon **name** and **munition** for which the threat falls within the **min** and **max** ranges will be chosen as the weapon and munition recommendation.

The **looking_tick_period** parameter specifies how often the task should process input when no targets are detected.

The **reevaluating_tick_period** parameter specifies how often the task should process input when a target is currently recommended.

The **recommendation_persistence_time** parameter specifies the amount of time that a chosen threat should remain chosen even after is becomes undetected by all available sensors.

The **assessment_type** specifies the type of algorithm used to determine threat. Currently, the only two types supported are are **air** and **ground**.

The **range_factor** parameter specifies a multiplier from 0.0 to 1.0 rating range as a criteria for threat with respect to other criteria. Currently, range is the only criteria used to classify threat.

The **aspect_factor** specifies a multiplier from 0.0 to 1.0 rating aspect angle as a criteria for threat with respect to other criteria. Currently, only assessment_type **air** uses this criteria.

The **persistence_factor** parameter specifies a multiplier by which an already chosen threat will be evaluated. For instance, a factor of 2.0 means that once a target is chosen, other targets will not be chosen unless they become less than half the distance to the vehicle than the chosen threat (assuming range is the only criteria).

The **mobility_and_fire_factor** specifies a multiplier from 0.0 to 1.0 rating the appearance of both a mobility kill and firepower kill for threat with respect to other criteria. Currently only assessment_type **ground** uses this criteria.

The **fire_factor** specifies a multiplier from 0.0 to 1.0 rating the appearance of just a firepower kill for threat with respect to other criteria. Currently only assessment_type **ground** uses this criteria.

The **max_hits_on_target** specifies the maximum number of shots hit on the target before the target will be considered not a threat (or undefeatable).

The **target_priorities** parameter contains the prioritized list of vehicle classes. Enemy vehicles will be compared to this list to determine the priority of the enemy vehicle. If an enemy does not match any class in the list then it will be given the lowest priority. The highest priority is 10 and the lowest is 1.

When a **SM_VAssess** task is created or modified, parameters in the parameter block of the task data structure are referenced to customize the task's behavior. Since libvassess implements a

vehicle level task, these parameters are typically initialized and modified by unit tasks which are responsible for directing vehicle level behavior.

The parameters are represented in the task data structure as follows:

```
typedef struct vassess_parameters
{
    ObjectID    sctr_rgt_bnd;
    ObjectID    sctr_lft_bnd;
    float64     max_threat_range;
    float64     max_threat_aspect;
    float64     min_threat_speed;
    float64     fire_at_pos[2];
    VASSESS_ROE permission;
    VASSESS_FIRE_TYPE fire_type;
} VASSESS_PARAMETERS;
```

**sctr_rgt_bnd** specifies the right sector boundary. Targets within a sector will have priority over targets outside of the sector.

**max_threat_range** specifies the left sector boundary. Targets within a sector will have priority over targets outside of the sector.

**max_threat_range** specifies the maximum range, in meters, beyond which a potential enemy will not be considered a threat.

**max_threat_aspect** specifies the maximum aspect angle, in radians, above which a potential enemy will not be considered a threat.

**min_threat_speed** specifies the minimum target velocity, in meters per second, below which a potential enemy will not be considered a threat.

**fire_at_position** is the position used when **permission** is FIRE_AT_POSITION.

**permission** specifies whether permission to fire is currently enabled. This value is supplied as a parameter which may be propagated as recommandations from this task (see Section 2.6 [vass'get'recommendation], page 9) should a potential enemy satisfy the target criterion specified by the **max_threat_range**, **max_threat_aspect** and **min_threat_speed** parameters. **permission** can take three values:

- VASSESS_HOLD_FIRE
- VASSESS_FIRE_AT_WILL
- VASSESS_FIRE_POSITION

`fire_type` specifies the method of firing at the enemy. `fire_type` can be set to:

- VASSESS_DISTRIBUTED_FIRE
- VASSESS_VOLLEY_FIRE
- VASSESS_NONE

If `fire_type` is set to VASSESS_DISTRIBUTED_FIRE then VAssess will try to choose a target that is not being targeted by someone else. If all spotted enemy vehicles are being targeted by someone else then the vehicle will target the highest priority enemy vehicle.

If `fire_type` is either VASSESS_VOLLEY_FIRE or VASSESS_NONE then VAssess will choose a target that is the highest priority. These two types do not check to see if the spotted enemy vehicle is being targeted by someone else.

# 2 Functions

The following sections describe each function provided by libvassess, including the format and meaning of its arguments, and the meaning of its return values (if any).

## 2.1 vass_init

```
void vass_init(data_path, reader_flags, tcc)
    char          *data_path;
    uint32         reader_flags;
    COORD_TCC_PTR tcc;
```

'data_path'
: Specifies the directory where data files are expected

'reader_flags'
: Specifies the flags to use when data files are read

'tcc'
: Specifies the local coordinate system

vass_init initializes libvassess. Call this before any other libvassess function.

## 2.2 vass_class_init

```
void vass_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
: Class of the parent (declared with class_declare_class)

vass_class_init creates a handle for attaching vassess class information to vehicles. The parent_class will likely be safobj_class.

## 2.3 vass_create

```
void vass_create(vehicle_id, params, ctdb, db, unit_entry)
```

```
int                          vehicle_id;
VASSESS_PARAMETRIC_DATA      *params;
CTDB                         *ctdb;
PO_DATABASE                  *db;
PO_DB_ENTRY                  *unit_entry;
```

'vehicle_id'
: Specifies the vehicle ID

'params'
: Specifies initial parameter values

'ctdb'
: Specifies terrain database information

'db'
: Specifies the PO database where the unit can be found

'unit_entry'
: Specifies the unit representing the vehicle in the PO database

**vass_create** creates the vassess class information for a vehicle and attaches it vehicle's block of libclass user data.

## 2.4 vass_destroy

```
void vass_destroy(vehicle_id, is_migration)
    int32 vehicle_id;
    int32 is_migration;
```

'vehicle_id'
: Specifies the vehicle ID

'is_migration'
: Specifies that the destroy is due to migration

**vass_destroy** frees the vassess class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

## 2.5 vass_init_task_state

```
void vass_init_task_state(task, state)
    TaskClass *task;
    TaskStateClass *state;
```

'task' Specifies a pointer to the task class object to be initialized.

'state' Returns the initialized state

Given a new SM_VAssess task that is about to be created, vass_init_task_state initializes the model size, and state variables.

## 2.6 vass_get_recommendation

```
void vass_get_recommendation(vehicle_id, rec)
    int32                    vehicle_id;
    VASSESS_RECOMMENDATION *rec;
```

'vehicle_id'
Specifies the vehicle ID

'rec' Specifies a pointer to return the recommendation into.

vass_get_recommendation returns the current recommendation for the highest threat. The recommendation has the following structure:

```
typedef struct vassess_recommendation
{
    int32       target;
    uint32      munition;
    char       *weapon;
    VASSESS_ROE permission;
} VASSESS_RECOMMENDATION;
```

target is interpreted as the the vehicle id of the highest threat.

munition is the recommended munition type to use against the threat.

weapon is the name of the weapon to use against the treat.

permission specifies the rules of engagement against the threat. It can take the following enumerated values:

'VASSESS_HOLD_FIRE'
This specifies not to shoot.

'VASSESS_FIRE_AT_WILL'
This specifies that permission to fire is granted.

The permission is derived from the initial permission as specified in the task's parameters. If the task parameter for permission is VASSESS_FIRE_AT_WILL, and if an available target satisfies all constraints specified in the other parameters, the recommandation for fire permission will be VASSESS_FIRE_AT_WILL. In all other cases, the recommendation will be VASSESS_HOLD_FIRE.

## 2.7 vass_get_recommendation_from_public

```
void vass_get_recommendation_from_public(db, task, rec)
    PO_DATABASE            *db;
    TaskClass              *task;
    VASSESS_RECOMMENDATION *rec;
```

'db' Specifies the PO database.

'task' Specifies a SM_VAssess task which is to be interpreted.

'rec' Specifies a pointer to return the recommendation into.

vass_get_recommendation_from_public returns the current recommendation for the highest threat. The values returned into rec are as in vass_get_recommendation (see Section 2.6 [vass'get'recommendation], page 9).

This function works on both locally simulated and remotely simulated vehicles, and may be used by user interface software to show the world from a vehicle's point of view.

## 2.8 vass_reset_recommendation

```
void vass_reset_recommendation(vehicle_id)
    int32 vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

vass_reset_recommendation causes the vassess task to re-analyze the threat situatation immediately on the next tick. This is typically used by a task which is using vassess recommendations

but discovers a new situation which should cause a new recommendation to be generated (such as destruction of previously recommended target).

# 3 Access Keys

In addition to the functions just described, libvassess also provides libaccess keys with which many variables can be fetched at once. These keys, and the type of argument they expect are given below:

***vass_recommendation***
**VASSESS_RECOMMENDATION *arg**

# LibVATAInt

# Table of Contents

# 1 Overview

Libvataint implements a vehicle-level task which controls the movement of a vehicle during an air-to-air intercept. In its current implementation, libvataint guides the aircraft on a pure-pursuit course of the enemy. The task state machine is written using the AAFSM format which is translated to C using the '`fsm2ch`' utility (see section 'Overview' in LibTask Programmer's Manual).

The states which comprise the intercept task are described below.

'`cant_intercept`'

This state is entered when the aircraft is on the ground and cannot perform an intercept. When in this state, the task continuously checks to see if the aircraft has taken off. Once the aircraft has taken off, the task transitions to the appropriate state to begin the intercept, based upon the current intercept geometry.

'`search_for_tgt`'

This state is entered when the aircraft does not detect the target on its radar. When in this state, the task steers the aircraft towards the last known enemy target position and continuously checks to see if it has acquired the target on its radar. Once the aircraft acquires the target, the task transitions to the appropriate state to begin or continue the intercept based upon the current intercept geometry.

'`analyze_geometry`'

This state is entered when the target is first detected at the start of an intercept. This state is used to simulate the time it takes for a pilot in a real aircraft to assess the intercept geometry. When in this state, the task steers the aircraft to point its nose at the target and maintain that course for a predetermined distance (specified in the task parametric data). After the aircraft has traveled that distance, the task computes the missiles it will shoot for this intercept, computes the distances at which to shoot those missiles, selects the first missile, and computes the desired target aspect and lateral separation to achieve an optimal positional advantage over the target. Depending upon the current intercept geometry, the task then immediately transitions to a state in which it prepares to take the first missile shot (if the enemy target is within desired shot range), or computes an initial maneuver which will put the aircraft on a course to achieve the desired target aspect or lateral separation and transitions to the initial_maneuver state to initiate that maneuver (if the enemy target is outside of desired shot range). If the enemy target drops off the aircraft's radar, the task transitions to the search_for_tgt state in an attempt to reacquire the target.

'`initial_maneuver`'

This state is entered when the enemy target is outside of the aircraft's desired shot range, and the aircraft therefore has time to make a maneuver in an attempt to achieve

a positional advantage over the enemy target. The initial maneuver is computed based upon a series of rules which take into account the current intercept geometry's target aspect, lateral separation, and altitude. When in this state, the task steers the aircraft in the direction of the computed initial maneuver and continuously checks to see if the desired target aspect has been achieved (and if so, transitions to collision_course state), if the enemy target is maneuvering (and if so, transitions to collision_course state), if the deisired lateral separation has been achieved (and if so, transitions to maintain_bogey_reciprocal state), or if the enemy target is getting close to the desired shot range (and if so, transitions to attack_heading state). If the enemy target drops off the aircraft's radar, the task transitions to the search_for_tgt state in an attempt to reacquire the target.

'maintain_bogey_reciprocal'

This state is entered when the aircraft is performing its initial maneuver and has achieved the desired lateral separation. When in this state, the task steers the aircraft on a course which is in the direction opposite the direction of travel of the enemy target (i.e. bogey heading reciprocal) and continuously checks to see if the desired target aspect has been achieved (and if so, transitions to collision_course state), if the enemy target is maneuvering (and if so, transitions to collision_course state), or if the enemy target is getting close to the desired shot range (and if so, transitions to attack_heading state). If the enemy target drops off the aircraft's radar, the task transitions to the search_for_tgt state in an attempt to reacquire the target.

'collision_course'

This state is entered whenever the desired target aspect has been achieved or the enemy target is maneuvering against the aircraft. When in this state, the task steers the aircraft on a collision course with the enemy target, and continuously checks to see if the enemy target is getting close to the desired shot range for its next missile shot (and if so, transitions to attack_heading state). If the enemy target drops off the aircraft's radar, the task transitions to the search_for_tgt state in an attempt to reacquire the target.

'attack_heading'

This state is entered when the aircraft is close to its desired shot range for its next missile shot. When in this state, the task steers the aircraft on an attack heading course (i.e. a course which is 1/3 the target aspect ahead of pure pursuit of the target) and continuously checks to see if the enemy target is within the desired shot range and within the selected missile's launch acceptability region (LAR) (and if so, transitions to shoot state). If the enemy target drops off the aircraft's radar, the task transitions to the search_for_tgt state in an attempt to reacquire the target.

'beam' This state is entered when the range to the enemy target has reached the range specified in the beam_range task parameter. The tactic of "turning into the beam" of the enemy aircraft's radar is used to defeat the enemy aircraft's pulse doppler radar modes, which

cannot track targets which are "in the beam". When in this state, the task steers the aircraft to a course which is 90 degrees from the enemy target's heading for a predetermined length of time.

'shoot' This state is entered when the aircraft is at the desired range to shoot the currently selected missile. When in this state, the task requests that a missile be fired by the vtargeter task (if fire permission is VATAINT_FIRE_AT_WILL). Regardless of whether or not a shot is taken, the task then checks to see if the crank task parameter was set to TRUE (and if so, transitions to the crank state), or if the crank task parameter was set to FALSE (and if so, steers the aircraft on its current course until it is time for the next shot, time to bugout, or time to merge).

'crank' This state is entered after the aircraft has shot a missile, if the crank task parameter was set to TRUE. When in this state, the task steers the aircraft on a course which is 40 degrees off of its attack heading course in the direction away from the enemy target and continuously checks to see if the enemy target is approaching the edge (< 15 degrees) of the radar scan volume (and if so, performs an EASY turn towards the enemy target to keep it safely inside the radar scan volume), if it is time to take another missile shot (and if so, transitions to attack_heading state), if it is time to go to the merge (and if so, transitions to merge state), or if it is time to bugout (and if so, transitions to bugout state). If the enemy target drops off the aircraft's radar, the task transitions to the search_for_tgt state in an attempt to reacquire the target.

'bugout' This state is entered after all planned missile shots have been taken and the aircraft is 12 nm from the enemy target and does not have a radar guided missile in flight. When in this state, the task steers the aircraft on a course which puts the enemy target 180 degrees behind the aircraft and continuously checks to see if the enemy target is 180 degrees behind the aircraft and the aircraft has created a separation of 6 nm from the enemy target (and if so, transitions to the END state to end the intercept).

'merge' This state is entered when the range between the aircraft and the enemy target is less than 12 nm, and either the aircraft has a radar guided missile in flight or the disengage_method task parameter was set to VATAINT_MERGE. When in this state, the task steers the aircraft on a pure pursuit course and continuously checks to see if the enemy target is getting close to the desired shot range for the next missile to shoot (and if so, transitions to shoot state), or if the range to the enemy target is less than 1 nm (and if so, transitions to the END state to end the intercept).

Libvataint depends on libvassess, libvtargeter, libvtab, libclass, libctdb, libpo, libhulls, libcomponents, libaccess, libreader, libstatmon, libeditor, libparmgr, libradar, librdrconst, libtime, libvecmat, libentity, libsensors, libsupplies, libfcs, liblar, and libtask.

## 1.1 Task Parameters

When a SM_VATAInt task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are represented in the task data structure as follows:

```
typedef struct vataint_parameters
{
    VehicleID                target_id;
    uint16                   padding1;
    float64                  target_bearing;  /* radians */
    float64                  target_range;    /* meters */
    VATAINT_FIRE_PERMISSION  fire_permission;
    int32                    weapon_count;
    VATAINT_WEAPONS_ENABLED  weapons_enabled[VATAINT_MAX_WEAPONS];
    int32                    crank;
    VATAINT_DISENGAGE_METHOD disengage_method;
    int32                    padding2;
    float64                  beam_range;
} VATAINT_PARAMETERS;
```

'target_id'
: Specifies the id of the vehicle to intercept.

'target_bearing'
: Specifies the bearing to the target in radians. This parameter is only set if the target_id is not known.

'target_range'
: Specifies the range to the target in meters. This parameter is only set if the target_id is not known.

'fire_permission'
: Specifies the fire permission (VATAINT_HOLD_FIRE, VATAINT_FIRE_AT_WILL) to be used during the intercept.

'weapon_count'
: Specifies the number of weapons in the weapons_enabled list.

'weapons_enabled'
: Specifies the weapons which the aircraft is allowed to shoot during the intercept.

'crank'
: Specifies whether to perform a crank maneuver after each shot taken during the intercept.

'disengage_method'
: Specifies how the aircraft should disengage from the target it is intercepting if it does not destroy it. This can take the values VATAINT_INTERNAL, VATAINT_MERGE, and VATAINT_BUGOUT. If it is set to VATAINT_INTERNAL, the disengage method

used will be based upon air-to-air intercept tactics taking into account the range to the target and whether a radar-guided missile is in flight. If it is set to VATAINT_MERGE, the aircraft will always go to the merge to disengage. If it is set to VATAINT_BUGOUT, the aircraft will always bugout (no later than 12 nm from the target) to disengage.

'beam_range'

Specifies the range in meters at which the aircraft should turn into the enemy target's radar beam.

# 2 Functions

The following sections describe each function provided by libvataint, including the format and meaning of its arguments, and the meaning of its return values (if any).

## 2.1 vataint_init

```
void vataint_init()
```

**vataint_init** initializes libvataint. Call this before any other libvataint function.

## 2.2 vataint_class_init

```
void vataint_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Class of the parent (declared with **class_declare_class**)

**vataint_class_init** creates a handle for attaching vataint class information to vehicles. The **parent_class** will likely be **safobj_class**.

## 2.3 vataint_create

```
void vataint_create(vehicle_id, params, po_db, ctdb)
    int32                    vehicle_id;
    VATAINT_PARAMETRIC_DATA *params;
    PO_DATABASE             *po_db;
    CTDB                    *ctdb;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'po_db' Specifies the PO database

'ctdb' Specifies the terrain database currently in use

**vataint_create** creates the vataint class information for a vehicle and attaches it vehicle's block of libclass user data.

## 2.4 vataint_destroy

```
void vataint_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

**vataint_destroy** frees the vataint class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

## 2.5 vataint_init_task_state

```
void vataint_init_task_state(task, state)
    TaskClass *task;
    TaskStateClass *state;
```

'task' Specifies a pointer to the task class object to be initialized.

'state' Returns the initialized state

Given a new **SM_VATAInt** task that is about to be created, **vataint_init_task_state** initializes the model size, and state variables.

## 2.6 vataint_able_to_intercept

```
int32 vataint_able_to_intercept(vehicle_id, ctdb)
    int32  vehicle_id;
    CTDB  *ctdb;
```

'vehicle_id'
Specified the Vehicle ID.

'ctdb' The ctdb pointer for terrain querries.

Determine if a vehicle is able to intercept. If the air vehicle is not in the air, then it is not able to intercept yet. This function will also be called by the unit air-to-air intercept task, in order to see if it needs to command a vehicle to take off before commanding it to intercept.

## 2.7 vataint_get_target_recommendation

```
void vataint_get_target_recommendation(vehicle_id, rec)
    int32                                vehicle_id;
    VTARGETER_TARGET_RECOMMENDATION     *rec;
```

'vehicle_id'
Specified the Vehicle ID.

'rec' The current target recomendation.

This function returs the current target recomendation that vataint is using.

**Libvatgrndtrgt**

# Table of Contents

# 1 Overview

TEMPLATE: Describe what this library does here.

## 1.1 Examples

TEMPLATE: Give examples here.

# 2 Functions

The following sections describe each function provided by libvatgrndtrgt, including the format and meaning of its arguments, and the meaning of its return values (if any).

TEMPLATE: Adjust alignment of descriptions

TEMPLATE: Correct argument lists and descriptions of these functions.

## 2.1 vatgtg_init

```
void vatgtg_init()
```

**vatgtg_init** initializes libvatgrndtrgt. Call this before any other libvatgrndtrgt function.

## 2.2 vatgtg_class_init

```
void vatgtg_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
: Class of the parent (declared with **class_declare_class**)

**vatgtg_class_init** creates a handle for attaching vatgrndtrgt class information to vehicles. The **parent_class** will likely be **safobj_class**.

## 2.3 vatgtg_create

```
void vatgtg_create(vehicle_id, params)
    int vehicle_id;
    VATGRNDTRGT_PARAMETRIC_DATA *params;
```

'vehicle_id'
: Specifies the vehicle ID

**'params'** Specifies initial parameter values

**vatgtg_create** creates the vatgrndtrgt class information for a vehicle and attaches it vehicle's block of libclass user data.

## 2.4 vatgtg_destroy

```
void vatgtg_destroy(vehicle_id)
    int vehicle_id;
```

**'vehicle_id'**
Specifies the vehicle ID

**vatgtg_destroy** frees the vatgrndtrgt class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

# LibVCAP

# Table of Contents

# 1 Overview

Libvcap implements a vehicle-level task which performs a Combat Air Patrol (CAP). It flies a racetrack pattern looking for targets to intercept. The task state machine is written using the AAFSM format which is translated to C using the '`fsm2ch`' utility (see section 'Overview' in LibTask Programmer's Manual).

Libvcap depends on libpo, libhulls, libvflwrte, libvtab, libclass, libctdb, libaccess, libstatmon, libeditor, libreader, libparmgr, librdrconst, libvecmat, libentity, libcomponents, and libtask.

## 1.1 Task Parameters

When a `SM_VCAP` task is created or modified, parameters in the parameter block of the task data structure are referenced. The parameters are described by the following structure:

```
typedef struct vcap_parameters
{
    ObjectID            position;
    uint16              _padding

    float64             orientation;
    float64             length_of_legs;
    float64             inbound_leg_speed;
    float64             outbound_leg_speed;
    float64             altitude;
} VCAP_PARAMETERS;
```

'`position`'
: A persistent object which defines the location to perform the CAP. This object can be a point object or a text object.

'`orientation`'
: Specifies the orientation of the racetrack pattern. This is used to determine the directions of the inbound and outbound legs.

'`length_of_legs`'
: Specifies how long each leg of the track should be.

'`inbound_leg_speed`'
: Specifies the speed that the aircraft will be moving on the inbound leg of the track.

'`outbound_leg_speed`'
: Specified the speed that the aircraft will be moving on the outbound leg of the track.

**'altitude'**

Specifies the altitude the aircraft will be flying at.

# 2 Functions

The following sections describe each function provided by libvcap, including the format and meaning of its arguments, and the meaning of its return values (if any).

## 2.1 vcap_init

```
void vcap_init()
```

**vcap_init** initializes libvcap. Call this before any other libvcap function.

## 2.2 vcap_class_init

```
void vcap_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Class of the parent (declared with **class_declare_class**)

**vcap_class_init** creates a handle for attaching vcap class information to vehicles. The **parent_class** will likely be **safobj_class**.

## 2.3 vcap_create

```
void vcap_create(vehicle_id, params, po_db, ctdb)
    int                    vehicle_id;
    VCAP_PARAMETRIC_DATA  *params;
    PO_DATABASE           *po_db;
    CTDB                  *ctdb;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'po_db' Specifies the PO database where the task can be found

'ctdb' Specifies the terrain database currently in use

**vcap_create** creates the vcap class information for a vehicle and attaches it vehicle's block of libclass user data.

## 2.4 vcap_destroy

```
void vcap_destroy(vehicle_id)
    int vehicle_id;
```

'vehicle_id'
Specifies the vehicle ID

**vcap_destroy** frees the vcap class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

## 2.5 vcap_init_task_state

```
void vcap_init_task_state(task, state)
    TaskClass *task;
    TaskStateClass *state;
```

'task' Specifies a pointer to the task class object to be initialized.

'state' Returns the initialized state

Given a new **SM_VCAP** task that is about to be created, **vcap_init_task_state** initializes the model size, and state variables.

## 2.6 vcap_able_to_cap

```
int32 vcap_able_to_cap(vehicle_id, ctdb)
    int32  vehicle_id;
    CTDB  *ctdb;
```

**'vehicle_id'**
Specifies the vehicle ID

**'ctdb'** Specifies the ctdb terrain database to use for terrain lookups

This routine determines if a vehicle is able to perform a CAP. If the air vehicle is not in the air, then it is not able to perform a CAP yet. This function will also be called by the unit CAP task, in order to see if it needs to command a vehicle to take off before commanding it to perform a CAP.

# LibVCollide

# Table of Contents

# 1 Overview

LibVCollide provides a task which allows vehicles to recover from collisions and near-collisions. The task has three states:

**waiting** In this state, the software waits for a collision to occur or for libmovemap to go into a state where it cannot plan (presumably because it has fallen into the configuration space boundaries of an obstacle).

**delay** After detecting the need to resolve a collision, the machine waits in this state for a random period of time, prior to dealing with the collision. The reason for this is two-fold: it simulates the delay a human would probably exhibit, and it prevents deadlock collision reactions between multiple vehicles by making the actions of each unique with respect to all others (analogous to the random delays between ethernet retransmissions after collisions).

**resolve_collision**
In this state, the machine instructs the movement arbitrator to move backward or forward, left or right, according to the nature of the collision, and the surrounding terrain features.

The format of the parametric data is as follows:

```
(SM_VCollide (background [ on | off ])
             (min_delay  <integer ms>)
             (max_delay  <integer ms>)
             (backup_distance <real meters>)
 )
```

**background** specifies whether the task should be automatically created as a background task of the vehicle when the vehicle is created.

**min_delay** and **max_delay** specify the range of delay times which should be generated for the random delaying state.

**backup_distance** specifies the distance which the vehicle should travel for each attempt to disengage from the collision.

## 1.1 Examples

The following is an example set of VCollide parameters, which yields delays between 2 and 5 seconds, and backs up 4 meters from each collision:

```
(SM_VCollide
 (background on)
 (min_delay 2000)
 (max_delay 5000)
 (backup_distance 4.0))
```

# 2 Functions

The following sections describe each function provided by libvcollide, including the format and meaning of its arguments, and the meaning of its return values (if any).

## 2.1 vcollide_init

```
void vcollide_init()
```

`vcollide_init` initializes libvcollide. Call this before any other libvcollide function.

## 2.2 vcollide_class_init

```
void vcollide_class_init(parent_class)
    CLASS_PTR parent_class;
```

'parent_class'
Class of the parent (declared with `class_declare_class`)

`vcollide_class_init` creates a handle for attaching vcollide class information to vehicles. The `parent_class` will likely be `safobj_class`.

## 2.3 vcollide_create

```
void vcollide_create(vehicle_id, params, db, unit_entry, ctdb)
    int                        vehicle_id;
    VCOLLIDE_PARAMETRIC_DATA   *params;
    PO_DATABASE                *db;
    PO_DB_ENTRY                *unit_entry;
    CTDB                       *ctdb;
```

'vehicle_id'
Specifies the vehicle ID

'params' Specifies initial parameter values

'db' Specifies the PO database

'unit_entry'
Specifies the PO entry of the unit which corresponds to this vehicle

'ctdb' Specifies the terrain database

**vcollide_create** creates the vcollide class information for a vehicle and attaches it vehicle's block of libclass user data. If the paramters so indicate, this routine will also create a task in the unit's background frame.

## 2.4 vcollide_destroy

```
void vcollide_destroy(vehicle_id, is_migration)
    int32 vehicle_id;
    int32 is_migration;
```

'vehicle_id'
Specifies the vehicle ID

'is_migration'
Specifies that the destroy is due to migration

**vcollide_destroy** frees the vcollide class information for a vehicle. This should be called before freeing the class user data with **class_free_user_data**.

## 2.5 vcollide_collision

```
void vcollide_collision(vehicle_id, with_whom)
    int32       vehicle_id;
    int32       with_whom;
```

'vehicle_id'
Specifies the vehicle ID

'with_whom'
Specifies the other party in the collision

**vcollide_collision** informs libvcollide that a collision occurred, so that the task may react to the collision. Collisions with the terrain should be indicated by a **with_whom** value of 0.

# Table of Contents

Naval Research Laboratory, Contract Number: N00014-92-C-2150
Data Item Number: A001, ModSAF B Software Documentation

# LibVecMat

## LibVecMat Programmer's Guide

Joshua E. Smith

$Revision: 1.25 $

# 1 Overview

Libvecmat is a vector and matrix operation library. Most functions are supported in many formats, and are named using a regular convention:

*2D or 3D* Name starts with `vmat2` or `vmat3`

*Arguments individually or in an array*
Next characters are `e_` or just `_`

*32 bit or 64 bit floating point*
Name ends with `32` or `64`

*Function or macro*
Name is in `lowercase` or `ALL CAPS`

For example, the 32 bit, 3D, vector add function which takes its arguments individually is called `vmat3e_vec_add64`. Most macro versions are not differentiated by bit length, because the compiler will know based upon the definitions of the variables used.

The manual entry for each function specifies which versions are provided. The prototype given in the manual entry uses generic terms as follows:

`scalar s` Either `float32 s` or `float64 s`

`vector v` One of the following:

```
float32 vx, vy
float32 vx, vy, vz
float64 vx, vy
float64 vx, vy, vz
float32 v[2]
float32 v[3]
float64 v[2]
float64 v[3]
```

`matrix m` One of the following:

```
float32 m[2][2]
float32 m[3][3]
float64 m[2][2]
float64 m[3][3]
```

Every function and macro has been written such that the same vector or matrix may be passed more than once. For example, the expression,

```
float64 a[3], b[3];
...
vmat3_cross_prod64(a, b, a);
```

is perfectly legal. However, the routines which take a combination of vectors and matrices do not check for a resultant vector being a row of a passed matrix. For example, the expression,

```
float64 a[3], b[3][3];
...
vmat3_vec_mat_mul64(a, b, b[1]);
```

will not work as intended.

Note that the obsolete library, libmatrix, was not consistent in its argument passing. Since libvecmat is consistent, you must be very careful when converting software to use libvecmat. Specifically, the functions which operate on scalars, and those which generate rotation matrices are very different.

# 2 Examples

Add two vectors, and normalize the result:

```
float64 vec[3];

VMAT3E_VEC_ADD(1.0, 2.0, 3.0,
               10.0, 20.0, 30.0,
               vec);
vmat3_unit64(vec, vec);
```